We claim:

Sub A1

1. A method of automatically analyzing at least one seedling germinated from at least one seed, comprising the steps of:

(a) capturing a digital image of the at least one seedling;

(b) identifying the at least one seedling in the captured digital image;

(c) determining a primary path of the at least one seedling;

(d) determining at least one value from the primary path of the at least one seedling; and

(e) determining a seed vigor index from at least the at least one value determined from the primary path of the at least one seedling.

2. The method of automatically analyzing at least one seedling according to claim 1:

(a) wherein said step of determining at least one value from the primary path of the at least one seedling comprises the step of determining a value corresponding to an overall length of the at least one seedling from the primary path of the at least one seedling; and

(b) wherein said step of determining a seed vigor index from at least the at least one value determined from the primary path of the at least one seedling comprises the step of determining a seed vigor index from at least the value corresponding to the overall length of the at least one seedling.

3. The method of automatically analyzing at least one seedling according to claim 1 further comprising the step of determining a separation point between the hypocotyl of the at least one seedling and the radicle of the at least one seedling; and:

(a) wherein said step of determining at least one value from the primary path of the at least one seedling comprises the step of determining a value corresponding to the length of at least one of the hypocotyl of the at least one seedling and the radicle of the at least one seedling; and

(b) wherein said step of determining a seed vigor index from at least the at least one value determined from the primary path of the at least one seedling comprises the step of determining a seed vigor index from at least the value corresponding to the length of at least one of the hypocotyl of the at least one seedling and the radicle of the at least one seedling.

Sub D1

4. The method of automatically analyzing at least one seedling according to claim 1 further comprising the step of determining a separation point between the hypocotyl of the at least one seedling and the radicle of the at least one seedling; and:

(a) wherein said step of determining at least one value from the primary path of the at least one seedling comprises the step of determining a hypocotyl length value corresponding to the length of the hypocotyl of the at least one seedling and a radicle length value corresponding to the length of the radicle of the at least one seedling; and

(b) wherein said step of determining a seed vigor index from at least the at least one value determined from the primary path of the at least one seedling comprises the step of determining a seed vigor index from at least the hypocotyl length value and the radicle length value.

5. The method of automatically analyzing at least one seedling according to claim 1 wherein said step of determining a primary path of the at least one seedling comprises the step of determining a locus of pixels, the locus of pixels corresponding to the primary path of the at least one seedling and the locus of pixels being narrower in width than the width of the at least one seedling in the digital image of the at least one seedling.

6. The method of automatically analyzing at least one seedling according to claim 1 wherein said step of determining a primary path of the at least one seedling comprises the step of determining a locus of pixels, the locus of pixels corresponding to the primary path of the at least one seedling and the locus of pixels being a predetermined number of pixels in width.

Sub A1

7. The method of automatically analyzing at least one seedling according to claim 1 wherein said step of determining a primary path of the at least one seedling comprises the step of determining a locus of pixels, the locus of pixels corresponding to the primary path of the at least one seedling and the locus of pixels being one pixel in width.

8. The method of automatically analyzing at least one seedling according to claim 1 further comprising the step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling, and

(a) wherein said step of determining a primary path of the at least one seedling comprises the step of determining a primary path for each of the separately identified overlapped seedlings;

(b) wherein said step of determining at least one value from the primary path of the at least one seedling comprises the step of determining from the primary path for each of the separately identified overlapped seedlings a value corresponding to an overall length of that separately identified overlapped seedling; and

(c) wherein said step of determining a seed vigor index from at least the at least one value determined from the primary path of the at least one seedling comprises the step of determining a seed vigor index from at least the plurality of values determined in step (b).

9. The method of automatically analyzing at least one seedling according to claim 1 further comprising the step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling, and

(a) wherein said step of determining a primary path of the at least one seedling comprises the step of determining a primary path for each of the separately identified overlapped seedlings;

Sub A1

(b) wherein said step of determining at least one value from the primary path of the at least one seedling comprises the step of determining from the primary path for each of the separately identified overlapped seedlings a value corresponding to the length of at least one of the hypocotyl of that separately identified overlapped seedling and the radicle of that separately identified overlapped seedling; and

(c) wherein said step of determining a seed vigor index from at least the at least one value determined from the primary path of the at least one seedling comprises the step of determining a seed vigor index from at least the plurality of values determined in step (b).

10. The method of automatically analyzing at least one seedling according to claim 1 further comprising the step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling, and

(a) wherein said step of determining a primary path of the at least one seedling comprises the step of determining a primary path for each of the separately identified overlapped seedlings;

(b) wherein said step of determining at least one value from the primary path of the at least one seedling comprises the step of determining from the primary path for each of the separately identified overlapped seedlings a hypocotyl length value corresponding to the length of the hypocotyl of that separately identified overlapped seedling and a radicle length value corresponding to the length of the radicle of that separately identified overlapped seedling; and

(c) wherein said step of determining a seed vigor index from at least the at least one value determined from the primary path of the at least one seedling comprises the step of determining a seed vigor index from at least the plurality of hypocotyl length values determined in step (b) and at least the plurality of radicle length values determined in step (b).

11. The method of automatically analyzing at least one seedling according to claim 8 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function.

12. The method of automatically analyzing at least one seedling according to claim 9 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function.

13. The method of automatically analyzing at least one seedling according to claim 10 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function.

14. The method of automatically analyzing at least one seedling according to claim 8 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function based on proposed configurations of at least partial primary paths of overlapped seedlings.

15. The method of automatically analyzing at least one seedling according to claim 9 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function based on proposed configurations of at least partial primary paths of overlapped seedlings.

16. The method of automatically analyzing at least one seedling according to claim 10 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function based on proposed configurations of at least partial primary paths of overlapped seedlings.

17. The method of automatically analyzing at least one seedling according to claim 8 wherein said step of separately identifying a plurality of overlapped seedlings in the digital

image of the at least one seedling comprises evaluating an energy function based on proposed configurations of at least partial primary paths of overlapped seedlings using the following heuristics: primary paths do not make unnaturally sharp turns and seedling edges should be used as much as possible.

Sub A1

18. The method of automatically analyzing at least one seedling according to claim 9 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function based on proposed configurations of at least partial primary paths of overlapped seedlings using the following heuristics: primary paths do not make unnaturally sharp turns and seedling edges should be used as much as possible.

19. The method of automatically analyzing at least one seedling according to claim 10 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function based on proposed configurations of at least partial primary paths of overlapped seedlings using the following heuristics: primary paths do not make unnaturally sharp turns and seedling edges should be used as much as possible.

20. The method of automatically analyzing at least one seedling according to claim 8 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function based on proposed configurations of at least partial primary paths of overlapped seedlings using the following heuristics: primary paths should not make unnaturally sharp turns, seedling edges should be used as much as possible, and all primary axes should have a hypocotyl/radicle separation point.

Sub A1

21. The method of automatically analyzing at least one seedling according to claim 9 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function based on proposed configurations of at least partial primary paths of overlapped seedlings using the following heuristics: primary paths should not make unnaturally sharp turns, seedling edges should be used as much as possible, and all primary axes should have a hypocotyl/radicle separation point.

22. The method of automatically analyzing at least one seedling according to claim 10 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function based on proposed configurations of at least partial primary paths of overlapped seedlings using the following heuristics: primary paths should not make unnaturally sharp turns, seedling edges should be used as much as possible, and all primary axes should have a hypocotyl/radicle separation point.

23. The method of automatically analyzing at least one seedling according to claim 1 further comprising the steps of determining a first locus of points indicating the hypocotyl of the at least one seedling, determining a second locus of points indicating the radicle of the at least one seedling, overlaying the first and second loci over an image of the seedlings to generate a composite image, and displaying the composite image.

24. The method of automatically analyzing at least one seedling according to claim 23 wherein said step of displaying the composite image comprises the step of displaying the composite image on a video display terminal.

Sub A1

25. The method of automatically analyzing at least one seedling according to claim 23 wherein said step of displaying the composite image comprises the step of printing the image on a printer or plotter.

26. A method of automatically analyzing at least one seedling germinated from at least one seed, comprising the steps of:

(a) capturing a digital image of the at least one seedling;

(b) determining a first locus of points indicating the hypocotyl of the at least one seedling;

(c) determining a second locus of points indicating the radicle of the at least one seedling;

(d) overlaying the first and second loci over an image of the seedlings to generate a composite image; and

(e) displaying the composite image.

27. A method of analyzing at least one seedling germinated from at least one seed, comprising the steps of:

(a) placing a growing medium in a shallow container;

(b) wetting the growing medium;

(c) placing the at least one seed onto the growing medium;

(d) germinating the at least one seed with the shallow container at an angle with respect to the vertical that is less than about 10°;

(e) capturing a digital image of the at least one seedling; and

(f) analyzing the captured digital image of the germinated seedling.

28. The method of analyzing at least one seedling according to claim 1 wherein said step of germinating the at least one seed with the shallow container at an angle with respect to the

vertical that is less than about 10° comprises the step of positioning the shallow container vertically.

29. The method of analyzing at least one seedling according to claim 1 wherein said step of capturing a digital image of the at least one seedling comprises capturing an image of the at least one seedling using a scanner having a scanner surface and positioned with its scanner surface oriented at least 90° from the horizontal.

30. The method of analyzing at least one seedling according to claim 1 wherein said step of capturing a digital image of the at least one seedling comprises capturing an image of the at least one seedling using a scanner having a scanner surface and positioned with its scanner surface substantially inverted so that it captures an image of at least one seedling positioned beneath the scanner.

```
// SeedlingAnalyzer.cpp: implementation of the SeedlingAnalyzer class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "Progeny.h"
#include "SeedlingAnalyzer.h"
#include "ProgenyDoc.h"
#include "ColorConversion.h"
#include "const.h"
#include <math.h>
#include <fstream.h>

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

#define GETRAND (rand()/(double)(RAND_MAX+1))

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

SeedlingAnalyzer::~SeedlingAnalyzer()
{

}

// Returns information needed to measure radicle and hypocotyl lengths on
// each seedling detected from a color image of seedlings.
std::vector<SeedlingInfo *> SeedlingAnalyzer::ProcessSeedlingImage(ImageIO* image)
{
      // Minimum pixel area for a blob to be qualified as a seedling blob
      const int c_minblobsize = 100;
      // kernel size used for median filtering
      const int c_medkernelsize = 3;
      // Minimum skeleton length to be qualified as a seedling blob
      const int c_minseglen = 15;
      // Minimum size for a blob to be considered as a cotyledon
      const int c_mincotsize = 10;
      // Minimum size for a blob to be considered as a seed coat
      const int c_mincoatsize = 10;

      // Seedling separation parameters
      const float c_angleW = 200.0f;
      const float c_initialTemperature = 2000.0f;
      const float c_lengthW = 0.0f;
      const int c_maxIteration = 50000;
      const float c_temperatureConst = 1.0f;
      const float c_unusedW = 10.0f;
      const float c_minEdgeLength = 10.0f;
      const float c_separationW = 200.0f;

      // Data structure to hold extract measurements of seedlings
      std::vector<SeedlingInfo *> seedlinginfo;
```

```
// origimg is the original RGB image of seedlings.
ExImage* origimg = (ExImage*)image;

// Create binary image of cotyledons (leaves)
ExImage* cotimg = origimg->Threshold(200, 255, 200, 255, 0, 200);

// Create binary image of seed coats
ExImage* coatimg = origimg->Threshold(60, 255, 0, 255, 0, 100);

// Binary threshold based on red channel
// To threshold, find the peak intensity on the red channel by FindPeakValue(0).
// The peak intensity + 40 is a good lower bound for thresholding.
ExImage* binimg = origimg->Threshold(origimg->FindPeakValue(0)+40, 255, 0, 255, 0,
255);

// Remove blobs of size less than c_minblobsize
BinaryObjects* binobjs = binimg->GetObjects();
delete binimg;
binobjs->FilterOut(c_minblobsize);
ExImage* cleanbinimg = binobjs->CreateImage();
delete binobjs;

// Perform median filtering to remove noise and smooth out edges

ExImage* smoothimg = cleanbinimg->MedianFilter(c_medkernelsize);
BinaryObjects* smoothobjs = smoothimg->GetObjects();

std::vector<ObjectInfo*> blobobjinfo = smoothobjs->GetObjectInfo();

// 4. Perform thinning (skeletonization)

ExImage* thinimg = smoothimg->Thin();
delete smoothimg;

// Throw out skeletons of length less than c_minsegln
BinaryObjects* binskel = thinimg->GetObjects();
delete thinimg;
binskel->FilterOut(c_minseglen);

// objinfo holds information for all detected objects in the
// skeletonized image.
std::vector<ObjectInfo*> objinfo = binskel->GetObjectInfo();

// Since the extracted skeleton contains root hairs and other
// noise, use the shortest path algorithm to find the primary
// axis.  Also, the second junction in the path is the hypocotyl
// separation, if it is not the end junction (i.e., not the bottom).

ShortestPathFinder spf;
ConnectivityGraph<Edge *>* jg;  // Junction graph.

bool added = false;  // True if seedlinginfo was updated.

// Perform the following loop for each seedling blob
for(i=0; i<objinfo.size(); i++)
{
```

```
            // Compute the junction graph for the seedling blob
            std::vector<Junction*> jlist;
            jg = binskel->CreateJunctionGraph(objinfo[i], jlist);

            // A seedling blob has to have more than 2 junctions
            // to be considered for further processing
            if(jlist.size() > 1)
            {
                  // Find the cotyledon junction.

                  // Find cotyledons and leaves in the current blob.

                  ExImage* cotblobimg = cotimg->Crop(max(0,objinfo[i]->xmin-8),
max(0,objinfo[i]->ymin-8),
                        min(origimg->GetWidth()-1,objinfo[i]->xmax+8), min(origimg-
>GetHeight()-1,objinfo[i]->ymax+8));

                  ExImage* coatblobimg = coatimg->Crop(max(0,objinfo[i]->xmin-8),
max(0,objinfo[i]->ymin-8),
                        min(origimg->GetWidth()-1,objinfo[i]->xmax+8), min(origimg-
>GetHeight()-1,objinfo[i]->ymax+8));

                  // Throw out blobs of size less than c_mincotsize
                  // cotinfo contains information for all extracted cotyledon
                  // blobs greater than or equal to c_mincotsize

                  BinaryObjects* cotobjs = cotblobimg->GetObjects();
                  delete cotblobimg;
                  cotobjs->FilterOut(c_mincotsize);
                  std::vector<ObjectInfo*> cotinfo = cotobjs->GetObjectInfo();

                  // Throw out blobs of size less than c_mincoatsize
                  // coatinfo contains information for all extracted seed coat
                  // blobs greater than or equal to c_mincoatsize

                  BinaryObjects* coatobjs = coatblobimg->GetObjects();
                  delete coatblobimg;
                  coatobjs->FilterOut(c_mincoatsize);
                  std::vector<ObjectInfo*> coatinfo = coatobjs->GetObjectInfo();

                  // coatindx holds id of seed coat junctions
                  std::vector<int> coatindx;

                  // Find the junction (in junction graph) that corresponds to seed coats
                  for(int c=0; c<coatinfo.size(); c++)
                  {
                        // Since objects were extracted from a cropped image, add offset
                        int coatx = max(0,objinfo[i]->xmin-8)+(coatinfo[c]->xmax +
coatinfo[c]->xmin)/2;
                        int coaty = max(0,objinfo[i]->ymin-8)+(coatinfo[c]->ymax +
coatinfo[c]->ymin)/2;

                        float mindist = 1000000.0f;
                        int minindx = -1;

                        // Loop through each junction in the junction graph to
                        // find the closest junction to the current seed coat
```

```
			for(int j=0; j<jlist.size(); j++)
			{
				Junction* junc = jlist[j];
				int dist = (coatx - junc->m_x) * (coatx - junc->m_x) +
(coaty - junc->m_y) * (coaty - junc->m_y);
				// the closest junction must be less than 10 pixels away
from the seed coat blob
				if((dist < mindist) && (dist < 10*10))
				{
					mindist = dist;
					minindx = j;
				}
			}
			// if such a seed coat is found, add it to coatindx
			if(minindx >= 0)
			{
				coatindx.push_back(minindx);
			}
		}


		// cotindx holds id of seed coat junctions.
		std::vector<int> cotindx;

		// Find the junction that corresponds to cotyledons.
		for(c=0; c<cotinfo.size(); c++)
		{
			cotindx.push_back(-1);

			// Since objects were extracted from a cropped image, add offset.
			int cotx = max(0,objinfo[i]->xmin-8)+(cotinfo[c]->xmax +
cotinfo[c]->xmin)/2;
			int coty = max(0,objinfo[i]->ymin-8)+(cotinfo[c]->ymax +
cotinfo[c]->ymin)/2;

			TRACE3("blob %d: cotx=%d, coty=%d\n", i, cotx, coty);

			float mindist = 1000000.0f;

			// Loop through each junction in the junction graph to
			// find the closest junction to the current cotyledon
			for(int j=0; j<jlist.size(); j++)
			{
				Junction* junc = jlist[j];
				int dist = (cotx - junc->m_x) * (cotx - junc->m_x) +  (coty
- junc->m_y) * (coty - junc->m_y);
				if(dist < mindist)
				{
					mindist = dist;
					cotindx[c] = j;
				}
			}
		}

		// Each seed coat that is close enough to a cotyledon is merged to the
cotyledon
		for(int cot=0; cot<cotindx.size(); cot++)
```

```
		{
			for(int coat=0; coat<coatindx.size(); coat++)
			{
				// if junctions are within 10 pixel distance, then drop the
coat junction.
				float sqdist = pow(jlist[cotindx[cot]]->m_x -
jlist[coatindx[coat]]->m_x, 2) +
					pow(jlist[cotindx[cot]]->m_y - jlist[coatindx[coat]]-
>m_y, 2);
				if(sqdist < 10.0f*10.0f)
				{
					coatindx.erase(coatindx.begin()+coat);
					coat--;
				}
			}
		}

		// Free memory
		for(int obj=0; obj<cotinfo.size(); obj++)
			delete cotinfo[obj];

		// Free memory
		for(obj=0; obj<coatinfo.size(); obj++)
			delete coatinfo[obj];

		// primaryAxes holds a primary axis for each detected seedling
		// in the current seedling blob
		std::vector<std::vector<int> > primaryAxes;

		// If no cotyledon/seed coat was detected,
		// assume the seedling blob contains only one seedling
		if((cotindx.size() == 0) && (coatindx.size() == 0))
		{
			// Find the longest shortest path to find the primary axis
			float maxdist = 0.0f;
			int maxindx1 = -1, maxindx2 = -1;
			for(int k=0; k<jg->GetSize(); k++)
			{
				std::vector<float> dist = spf.FindShortestPathDistance(*jg,
k);

				for(int m=0; m<dist.size(); m++)
				{
					if(maxdist < dist[m])
					{
						maxdist = dist[m];
						maxindx1 = k;
						maxindx2 = m;
					}
				}
			}
			ASSERT((maxindx1 != -1) && (maxindx2 != -1));
			// Since no information is given on which end junction belongs
			// to a cotyledon, find out which index is top/bottom.
			// Assume the top junction belongs to a cotyledon.
			int y1 = jlist[maxindx1]->m_y;
			int y2 = jlist[maxindx2]->m_y;
```

```
			if(y1 > y2)
			{
				int temp = maxindx1;
				maxindx1 = maxindx2;
				maxindx2 = temp;
			}

			primaryAxes.push_back(spf.FindShortestPath(*jg, maxindx1,
maxindx2));
		}
		// If a cotyledon was detected but no seed coat, find the shortest
		// path from the cotyledon junction to every other junciton,
		// and take the longest path as the primary axis
		else if((cotindx.size() == 1)&&(coatindx.size()==0))
		{
				std::vector<float> dist = spf.FindShortestPathDistance(*jg,
cotindx[0]);

				int maxindx = -1;
			    float maxdist = -1.0f;

				for(int m=0; m<dist.size(); m++)
				{
					if(maxdist < dist[m])
					{
						maxdist = dist[m];
						maxindx = m;
					}
				}

				ASSERT(maxindx >= 0);

			int y1 = jlist[cotindx[0]]->m_y;
			int y2 = jlist[maxindx]->m_y;

			int indx1, indx2;

			if(y1 > y2)
			{
				indx1 = maxindx;
				indx2 = cotindx[0];
			}
			else
			{
				indx1 = cotindx[0];
				indx2 = maxindx;
			}

			primaryAxes.push_back(spf.FindShortestPath(*jg, indx1, indx2));
		}
		// If there is one seed coat and no cotyledon, assume one seedling
		// within the blob
		else if((coatindx.size() == 1)&&(cotindx.size()==0))
		{
			// Find the longest shortest path.
				std::vector<float> dist = spf.FindShortestPathDistance(*jg,
coatindx[0]);
```

```
                        int maxindx = -1;
                      float maxdist = -1.0f;

                        for(int m=0; m<dist.size(); m++)
                        {
                              if(maxdist < dist[m])
                              {
                                    maxdist = dist[m];
                                    maxindx = m;
                              }
                        }
                        ASSERT(maxindx >= 0);

                  int y1 = jlist[coatindx[0]]->m_y;
                  int y2 = jlist[maxindx]->m_y;

                  int indx1, indx2;

                  if(y1 > y2)
                  {
                        indx1 = maxindx;
                        indx2 = coatindx[0];
                  }
                  else
                  {
                        indx1 = coatindx[0];
                        indx2 = maxindx;
                  }

                  primaryAxes.push_back(spf.FindShortestPath(*jg, indx1, indx2));
            }
            // This is the case where there are multiple cotyledons in the blob
            else if(coatindx.size() + cotindx.size() > 1)
            {
                  std::vector<int> startjuncs;
                  for(int c=0; c<coatindx.size(); c++)
                        startjuncs.push_back(coatindx[c]);
                  for(c=0; c<cotindx.size(); c++)
                        startjuncs.push_back(cotindx[c]);
                  // Obtain primary axis for each seedling in the blob
                  primaryAxes = SeparateSeedlings(jg, startjuncs, c_maxIteration,
c_lengthW, c_angleW, c_unusedW, c_initialTemperature, c_temperatureConst,
c_minEdgeLength, c_separationW);
            }

            // Perform hypocotyl/radicle separation on each seedling found in the
blob
            // The first junction encountered while traversing the primary path
from the
            // cotyledon junction that separates the seedling into hypocotyl and
radicle such that
            // hypocotyl : radicle length ratio is no less than 0.15
            // is the separation point

            for(int s=0; s<primaryAxes.size(); s++)
            {
```

```
                    if(primaryAxes[s].size() > 1)
                    {
                          int k = 1;

                          SeedlingInfo* sinfo = new SeedlingInfo;
                          sinfo->hypocotyl_length = jg->GetEdge(primaryAxes[s][k-1],
primaryAxes[s][k])->m_length;
                          sinfo->radicle_length = 0;

                          for(int j=1; j<primaryAxes[s].size()-1; j++)
                          {
                                sinfo->radicle_length += jg-
>GetEdge(primaryAxes[s][j], primaryAxes[s][j+1])->m_length;
                          }

                          // Check to see if hypocotyl/radicle ratio is not so
extreme.
                          // If so, extend hypocotyl and shorten radicle.
                          k++;
                          while(((float)sinfo->hypocotyl_length / sinfo-
>radicle_length < 0.15) && (primaryAxes[s].size() > k))
                          {
                                sinfo->hypocotyl_length += jg-
>GetEdge(primaryAxes[s][k-1], primaryAxes[s][k])->m_length;
                                sinfo->radicle_length -= jg-
>GetEdge(primaryAxes[s][k-1], primaryAxes[s][k])->m_length;
                                k++;
                          }

                          sinfo->hyporad_separation = k - 1;
                          sinfo->primary_axis = primaryAxes[s];
                          sinfo->junction_graph = jg;

                          // Compute the bounding box for the seedling

                          int xmin = 1000000, xmax = -1, ymin = 1000000, ymax = -1;
                          for(int m=0; m<primaryAxes[s].size()-1; m++)
                          {
                                Edge* edge = jg->GetEdge(primaryAxes[s][m],
primaryAxes[s][m+1]);

                                if(edge->m_xmin < xmin)
                                      xmin = edge->m_xmin;
                                if(edge->m_xmax > xmax)
                                      xmax = edge->m_xmax;
                                if(edge->m_ymin < ymin)
                                      ymin = edge->m_ymin;
                                if(edge->m_ymax > ymax)
                                      ymax = edge->m_ymax;
                          }

                          sinfo->topleft.x = xmin;
                          sinfo->topleft.y = ymin;
                          sinfo->bottomright.x = xmax;
                          sinfo->bottomright.y = ymax;

                          seedlinginfo.push_back(sinfo);
```

```
                            added = true;
                        }
                }  // if there is one cotyledon in the blob
        }  // if there are more than one junctions
        if(!added)
                delete jg;
        else
                added = false;
    }

    // In case cotyledon/hypocotyl separation was undetected,
    // assume separation point at the mean ratio.

    for(int i=0; i<seedlinginfo.size(); i++)
    {
        if(seedlinginfo[i]->hyporad_separation == 0)
        {
                seedlinginfo[i]->useAverageSeparation = true;
        }
    }

    // Free memory.

    delete cotimg;
    delete coatimg;

    return seedlinginfo;
}

// Separate seedlings using simulated annealing.
std::vector<std::vector<int> > SeedlingAnalyzer::SeparateSeedlings(ConnectivityGraph<Edge
>* jgraph, std::vector<int> startjunc,
int loopmax, float lengthW, float angleW, float unusedW, float temperature, float
temperatureConst, float minEdgeLength, float separationW)
{
    // start contains the ID of junctions that are preassigned to seedlings.
    // start.size() is the number of seedlings assigned initially.

    // Keep track of the path for each seedling.
    std::vector<std::vector<int> > paths;

    // Keep track of the length for the paths.
    std::vector<float> pathlength;

    // Keep track of the last "valid" end angle taken for each seedling.
    std::vector<std::vector<float> > endAngle;

    // Keep track of hypocotyl/radicle separation
    std::vector<int> RHseparation;
    std::vector<int> RHseparation2;
    std::vector<float> separationlen;

    // Initialize data structures by starting out with a single
    // junction for each seedling
    for(int i=0; i<startjunc.size(); i++)
    {
        std::vector<int> path;
```

```
        paths.push_back(path);
        paths[i].push_back(startjunc[i]);

        std::vector<float> list;
        endAngle.push_back(list);

        pathlength.push_back(0.0f);
        RHseparation.push_back(-1);  // Initially, paths don't have RH separation
        RHseparation2.push_back(-1);
        separationlen.push_back(0.0f);
    }


    // Keep track of which edge is occupied

    std::vector<std::set<int> > edgeOccupation;
    for(i=0; i<jgraph->GetNumEdges(); i++)
    {
        std::set<int> edgeset;
        edgeOccupation.push_back(edgeset);
    }

    // Set initial temperature for annealing.

    float energy = 0.0f;

    // Compute the energy of the configuration.
    // Since no edges are occupied by seedlings yet,
    // the energy is the sum of penalties for
    // unoccupied edges.
    for(i=0; i<jgraph->GetSize(); i++)
    {
        for(int j=i+1; j<jgraph->GetSize(); j++)
        {
            if(jgraph->IsEdge(i, j))
                energy += unusedW * jgraph->GetEdge(i, j)->m_length;
        }
    }

    // add energy for not having hypocotyl/radicle separation.
    energy += separationW * startjunc.size();

    for(i=0; i<loopmax; i++)
    {
        // Choose which seedling to update.
        int seedID = (int)(GETRAND*(double)startjunc.size());
        int endjunc = paths[seedID].back();

        bool changeAngle = false;  // Whether to change the last "valid" angle or
not.

        // Extend end or remove end.
        // Choose an edge by throwing a die.

        // Copy all neighboring junctions to neighbors.
        std::vector<int> neighbors;
        for(int j=0; j<jgraph->GetSize(); j++)
```

```
        {
              if(jgraph->IsEdge(endjunc, j))
                    neighbors.push_back(j);
        }
        int choice = neighbors[(int)(GETRAND*(double)neighbors.size())];

        // Try to remove edge if choice is the junction one before the
        // current junction.
        if((paths[seedID].size() > 1) && (choice == *(paths[seedID].end()-2)))
        {
              // Remove edge --- compute delta energy.
              // *-------> ... *------->*
              // start         choice    end

              float deltaEnergy;
              float edgeLength = jgraph->GetEdge(choice, endjunc)->m_length;

              // Removing hypo/rad separation increases energy
              if(RHseparation2[seedID] == choice)
              {
                    deltaEnergy += separationW;
              }


              // If the edge is sufficiently long, subtract energy for the angle
              if(edgeLength > minEdgeLength)
              {
                    if(endAngle[seedID].size() > 1)
                    {
                          float newangle = jgraph->GetEdge(choice, endjunc)-
>GetAngle(choice);
                          float angle = ComputeAngle(*(endAngle[seedID].end()-2),
newangle);
                          deltaEnergy -= angleW * angle * angle;
                    }
                    changeAngle = true;
              }

              // If the edge is no longer occupied after removal, increase
              // energy.
              if((edgeOccupation[jgraph->GetEdgeID(choice, endjunc)].size()==1) &&
                 (*edgeOccupation[jgraph->GetEdgeID(choice, endjunc)].begin() ==
seedID))
                    deltaEnergy += unusedW * edgeLength;

              if((deltaEnergy < 0.0f) || (exp(-deltaEnergy/(temperatureConst*
temperature))) > GETRAND)
              {
                    // Remove the end junction.
                    ASSERT(!paths[seedID].empty());
                    edgeOccupation[jgraph->GetEdgeID(choice, endjunc)].erase(seedID);
                    paths[seedID].pop_back();

                    if(changeAngle)
                    {
                          ASSERT(endAngle.size() > 0);
                          endAngle[seedID].pop_back();
```

```
                    }

                    // if the chosen junction was the hypo/rad separation for
                    // the seedling, unmark the separation.
                    if(RHseparation[seedID] == choice)
                    {
                         RHseparation[seedID] = -1;
                    }

                    if(RHseparation2[seedID] == choice)
                    {
                         RHseparation2[seedID] = -1;
                    }

                    pathlength[seedID] -= edgeLength;

                    energy += deltaEnergy;
               }
          }
          // Add edge if the selected junction is not the end junction itself
          else if((choice != endjunc) && (edgeOccupation[jgraph->GetEdgeID(endjunc,
choice)].find(seedID) == edgeOccupation[jgraph->GetEdgeID(endjunc, choice)].end()))
          {
               // Add edge
               // *-------> ... *------->*
               // start         end     choice

               ASSERT(jgraph->IsEdge(endjunc, choice));

               bool newseparation = false;  // Whether choice adds a hypo/rad
separation or not.
               bool separationcomplete = false;

               float deltaEnergy;
               float edgeLength = jgraph->GetEdge(endjunc, choice)->m_length;
               // If the edge is not already occupied, the energy goes down.
               if(edgeOccupation[jgraph->GetEdgeID(endjunc, choice)].empty())
                    deltaEnergy -= unusedW * edgeLength;

               int numneighbors = 0;
               // See if choice is a separation point by checking for a
               // neighbor edge that is short (and has a degree one at the other
end??)
               for(int j=0; j<jgraph->GetSize(); j++)
               {
                    if(jgraph->IsEdge(choice, j))
                    {
                         numneighbors++;
                    }
               }

               // Mark if this is a new hypo/rad separation.
               if(RHseparation[seedID] == -1)
               {
                    if((numneighbors > 2) && (pathlength[seedID] > 20.0f))
                    {
                         newseparation = true;
```

```
				}
			}
			else if((RHseparation2[seedID] == -1) && (pathlength[seedID] -
separationlen[seedID] > 20.0f))
			{
				deltaEnergy -= separationW;
				separationcomplete = true;
			}


			if(edgeLength > minEdgeLength)
			{
				if(!endAngle[seedID].empty())
				{
					float newangle = jgraph->GetEdge(endjunc, choice)-
>GetAngle(endjunc);
					float angle = ComputeAngle(endAngle[seedID].front(),
newangle);
					deltaEnergy += angleW * angle * angle;
					// egraph->GetEdge(jgraph-
>GetEdgeID(paths[seedID][paths[seedID].size()-2], endjunc), jgraph->GetEdgeID(endjunc,
choice));
				}
				changeAngle = true;
			}

			if((deltaEnergy < 0.0f) || (exp(-deltaEnergy/(temperatureConst*
temperature))) > GETRAND)
			{
				// mark that this seedling has occupied the edge
				edgeOccupation[jgraph->GetEdgeID(endjunc,
choice)].insert(seedID);
				ASSERT(edgeOccupation[jgraph->GetEdgeID(endjunc, choice)].size()
<= startjunc.size());
				paths[seedID].push_back(choice);

				if(changeAngle)
					endAngle[seedID].push_back(jgraph->GetEdge(endjunc,
choice)->GetAngle(choice));

				pathlength[seedID] += edgeLength;

				if(newseparation)
				{
					RHseparation[seedID] = choice;
					separationlen[seedID] = pathlength[seedID];
				}

				if(separationcomplete)
					RHseparation2[seedID] = choice;

				energy += deltaEnergy;
			}
		}

		// Adjust temperature.
		temperature *= 0.99f;
```

```
    } // Simulated annealing loop.

    return paths;
}

float SeedlingAnalyzer::ComputeAngle(float lastAngle, float newAngle)
{
    // Compute the angle between lastAngle and newAngle.

    float angle = fabs(newAngle - lastAngle);
    if(angle > PI)
        angle = DPI - angle;

    return (PI-angle);
}
```

```
// File: Image.cpp
// Author: Yusaku Sako
// Date: 07/11/99

#include "stdafx.h"  // for Windows
#include "Image.h"
#include "ImageThinning.h"  // for thinning a binary image
#include "ConnectivityGraph.h"
#include "Const.h"
#include <math.h>

#ifndef ROUND
#define ROUND(x)  ((int)(x+0.5))
#endif


ExImage* ExImage::Crop(int lx, int ly, int rx, int ry)
{
	Image* dupimage = duplicate_Image(m_image);
	Image* newimage = ::crop(dupimage, ly, lx, ry-ly+1, rx-lx+1);

	ExImage* returnimage = new ExImage;
	returnimage->m_image = newimage;

	return returnimage;
}

ExImage* ExImage::TransformToHSV(int numlevels)
{
	ExImage* newimg = (ExImage*)GetCopy();

	float norm[3];
	norm[0] = numlevels;  norm[1] = numlevels;  norm[2] = numlevels;
	newimg->m_image = colorxform(newimg->m_image, HSV, norm, NULL, 1);

	return newimg;
}

ExImage* ExImage::Get2DHistogram(int xcolor, int ycolor, int size)
{
	ExImage* histImg = new ExImage;
	histImg->m_image = new_Image(PGM, GRAY_SCALE, 1, size,
		size, CVIP_BYTE, REAL);

	const int HIGH = 255;
	const int BIAS = 128;
	unsigned char ** xpix = (unsigned char**)m_image->image_ptr[xcolor]->rptr;
	unsigned char ** ypix = (unsigned char**)m_image->image_ptr[ycolor]->rptr;
	unsigned char ** hpix = (unsigned char**)histImg->m_image->image_ptr[0]->rptr;

	int kx, ky, hx, hy, max, kk;

	// Clear image.
	for(int i=0; i<size; i++)
		for(int j=0; j<size; j++)
		{
			hpix[i][j] = (unsigned char)0;
```

```
        }

    max = 0;
    kx = 1;
    ky = 1;
    for(i=0; i<size; i++)
    {
        for(int j=0; j<size; j++)
        {
            hy = (HIGH - ypix[i][j])/ky;
            hx = (xpix[i][j])/kx;
            if(hpix[hy][hx] < HIGH)
                hpix[hy][hx]++;
            if(max < hpix[hy][hx])
                max = hpix[hy][hx];
        }
    }
    for(i=0; i<size; i++)
    {
        for(int j=0; j<size; j++)
        {
            if(hpix[i][j] !=0)
            {
                kk = hpix[i][j]*HIGH/max+BIAS;
                if(kk > HIGH)
                    hpix[i][j] = (unsigned char)HIGH;
                else
                    hpix[i][j] = (unsigned char)kk;
            }
        }
    }
    return histImg;


ExImage* ExImage::Threshold(unsigned char thresh)
{
    ExImage *returnImage = new ExImage;

    returnImage->m_image = threshold_segment(m_image, thresh, CVIP_NO);
 /*
 <inputImage> - pointer to Image structure
 <threshval> - threshold value
 <thresh_inbyte>
          - CVIP_NO apply threshval directly to image data;
          - CVIP_YES  threshval is CVIP_BYTE range; remap to image
                data range before thresholding.
 */
    return returnImage;
}

ExImage* ExImage::Threshold(int rlow, int rhigh, int glow, int ghigh, int blow, int
bhigh)
{
    ExImage* threshImg = new ExImage;
    threshImg->m_image = new_Image(PGM, GRAY_SCALE, 1, getNoOfRows_Image(m_image),
         getNoOfCols_Image(m_image), CVIP_BYTE, REAL);
```

```
	unsigned char** origR = (unsigned char**)m_image->image_ptr[0]->rptr;
	unsigned char** origG = (unsigned char**)m_image->image_ptr[1]->rptr;
	unsigned char** origB = (unsigned char**)m_image->image_ptr[2]->rptr;

	unsigned char** dest = (unsigned char**)threshImg->m_image->image_ptr[0]->rptr;

	for(int y=0; y<getNoOfRows_Image(m_image); y++)
		for(int x=0; x<getNoOfCols_Image(m_image); x++)
		{
			bool flag = true;

			if(origR[y][x] < rlow || origR[y][x] > rhigh)
				flag = false;
			if(origG[y][x] < glow || origG[y][x] > ghigh)
				flag = false;
			if(origB[y][x] < blow || origB[y][x] > bhigh)
				flag = false;

			if(flag)
				dest[y][x] = 255;
			else
				dest[y][x] = 0;
		}

	return threshImg;


ExImage* ExImage::Threshold(ThreshParams thresh)
{
	return Threshold(thresh.rmin, thresh.rmax, thresh.gmin, thresh.gmax, thresh.bmin,
thresh.bmax);


// Perform Thresholding segmentation based on histogram
ExImage* ExImage::HistogramThreshold()
{
	ExImage *returnImage = new ExImage;
	returnImage->m_image = hist_thresh_gray(m_image);
	return returnImage;
}

ExImage* ExImage::HistogramEqualization()
{
	ExImage *returnImage = new ExImage;
	Image* temp;

	temp = histeq(m_image, 0);
	//temp = histeq(temp, 1);
	//temp = histeq(temp, 2);
	returnImage->m_image = temp;
	returnImage->m_image = remap_Image(temp, CVIP_BYTE, 0, 255);

	return returnImage;
}

ExImage* ExImage::EdgeDetect(int type)
{
```

```
	ExImage *returnImage = new ExImage;
	returnImage->m_image = edge_detect_setup(m_image, 1);
	return returnImage;
}

// Perform skeletonization on the image.
ExImage* ExImage::Thin()
{
	ExImage *returnImage = new ExImage;
	returnImage->m_image = ::ImageThinning(m_image);
	return returnImage;
}

ExImage* ExImage::MorphOpen(int kerneltype, int kernelsize,
		int kernelheight, int kernelwidth)
{
	ExImage *returnImage = new ExImage;
	returnImage->m_image = ::MorphOpen(m_image, kerneltype, kernelsize, kernelheight,
kernelwidth);
	// DEBUG
	CString msg;
	msg.Format("Image params: width=%d height=%d bands=%d colorspace=%d", returnImage-
>m_image->image_ptr[0]->cols, returnImage->m_image->image_ptr[0]->rows, returnImage-
>m_image->bands, returnImage->m_image->color_space);
	//AfxMessageBox(msg);
	return returnImage;
}

ExImage* ExImage::MorphClose(int kerneltype, int kernelsize,
		int kernelheight, int kernelwidth)
{
	ExImage *returnImage = new ExImage;
	returnImage->m_image = ::MorphClose(m_image, kerneltype, kernelsize, kernelheight,
kernelwidth);
	return returnImage;
}

ExImage* ExImage::MorphDilate(int kerneltype, int kernelsize,
		int kernelheight, int kernelwidth)
{
	ExImage *returnImage = new ExImage;
	returnImage->m_image = ::MorphDilate(m_image, kerneltype, kernelsize, kernelheight,
kernelwidth);
	return returnImage;
}

ExImage* ExImage::MorphErode(int kerneltype, int kernelsize,
		int kernelheight, int kernelwidth)
{
	ExImage *returnImage = new ExImage;
	returnImage->m_image = ::MorphErode(m_image, kerneltype, kernelsize, kernelheight,
kernelwidth);
	return returnImage;
}

// Convert a color image to gray scale based on luminance.
ExImage* ExImage::ConvertToGrayscale(int maxvalue)
```

```
{
      ExImage *returnImage = new ExImage;
      returnImage->m_image = ::CVIPluminance(m_image, maxvalue,
     CVIP_YES, CVIP_NO);
      return returnImage;
}

// Perform median filter on the image.
ExImage* ExImage::MedianFilter(int kernelsize)
{
      ExImage *returnImage = new ExImage;
      returnImage->m_image = ::median_filter(m_image, kernelsize);
      return returnImage;
}

ExImage* ExImage::CutOut(int lx, int ly, int rx, int ry)
{
      ASSERT(lx>=0);
      ASSERT(ly>=0);
      ASSERT(rx<getNoOfCols_Image(m_image));
      ASSERT(ry<getNoOfRows_Image(m_image));

      Image* dupimg = duplicate_Image(m_image);
      unsigned char** pix = (unsigned char**)dupimg->image_ptr[0]->rptr;

      for(int y=ly; y<=ry; y++)
      {
            for(int x=lx; x<=rx; x++)
            {
                  pix[y][x] = 0U;
            }
      }

      ExImage *returnImage = new ExImage;
      returnImage->m_image = dupimg;
      return returnImage;
}

ExImage* ExImage::Minus(ExImage* inimage, int lx, int ly, int rx, int ry)
{
      Image* diffimg = duplicate_Image(m_image);

      unsigned char** pix1 = (unsigned char**)diffimg->image_ptr[0]->rptr;
      unsigned char** pix2 = (unsigned char**)inimage->m_image->image_ptr[0]->rptr;

      for(int y=ly; y<=ry; y++)
      {
            for(int x=lx; x<=rx; x++)
            {
                  int diff = pix1[y][x] - pix2[y-ly][x-lx];
                  diff = (diff<0) ? 0 : diff;
                  pix1[y][x] = diff;
            }
      }

      ExImage* returnImage = new ExImage;
      returnImage->m_image = diffimg;
```

```
	return returnImage;
}

ExImage* ExImage::Blend(ExImage* inimage, float weight_orig, float weight_in, bool
invert_orig, bool invert_in)
{
	// Image sizes must match.
	ASSERT(getNoOfCols_Image(m_image) == getNoOfCols_Image(inimage->m_image));
	ASSERT(getNoOfRows_Image(m_image) == getNoOfRows_Image(inimage->m_image));
	ASSERT((weight_orig >= 0.0) && ( weight_orig <= 1.0));

	Image* newimage = new_Image(PPM, RGB, 3, getNoOfRows_Image(m_image),
		getNoOfCols_Image(m_image),CVIP_BYTE, REAL);

	int numbands = getNoOfBands_Image(inimage->m_image);

	char **src1R = m_image->image_ptr[0]->rptr;
	char **src2R = inimage->m_image->image_ptr[0]->rptr;
	unsigned char **destR = (unsigned char**)newimage->image_ptr[0]->rptr;
	char **src1G = m_image->image_ptr[1]->rptr;
	char **src2G = (numbands>1) ? inimage->m_image->image_ptr[1]->rptr : inimage-
>m_image->image_ptr[0]->rptr;
	unsigned char **destG = (unsigned char**)newimage->image_ptr[1]->rptr;
	char **src1B = m_image->image_ptr[2]->rptr;
	char **src2B = (numbands>2) ? inimage->m_image->image_ptr[2]->rptr : inimage-
>m_image->image_ptr[0]->rptr;
	unsigned char **destB = (unsigned char**)newimage->image_ptr[2]->rptr;

	unsigned char src1r, src2r, src1g, src2g, src1b, src2b;

	for(int y=0; y<getNoOfRows_Image(m_image); y++)
		for(int x=0; x<getNoOfCols_Image(m_image); x++)
		{
			if(invert_orig)
			{
				src1r = -src1R[y][x]-1;
				src1g = -src1G[y][x]-1;
				src1b = -src1B[y][x]-1;
			}
			else
			{
				src1r = src1R[y][x];
				src1g = src1G[y][x];
				src1b = src1B[y][x];
			}
			if(invert_in)
			{
				src2r = -src2R[y][x]-1;
				src2g = -src2G[y][x]-1;
				src2b = -src2B[y][x]-1;
			}
			else
			{
				src2r = src2R[y][x];
				src2g = src2G[y][x];
				src2b = src2B[y][x];
			}
```

```
                destR[y][x]                                    (unsigned
char)((weight_orig*(float)(src1r)+weight_in*(float)src2r)/2);
                destG[y][x]                                    (unsigned
char)((weight_orig*(float)(src1g)+weight_in*(float)src2g)/2);
                destB[y][x]                                    (unsigned
char)((weight_orig*(float)(src1b)+weight_in*(float)src2b)/2);
            }

    ExImage *ret = new ExImage;
    ret->m_image = newimage;
    return ret;
}

ExImage* ExImage::Mask(ExImage* maskimage, ColorType bgcolor)
{
    ExImage * outimage = (ExImage*)GetCopy();

    unsigned char** mpix = maskimage->GetPixelArray(0);
    unsigned char** srcR = GetPixelArray(0);
    unsigned char** srcG = GetPixelArray(1);
    unsigned char** srcB = GetPixelArray(2);
    unsigned char** dstR = outimage->GetPixelArray(0);
    unsigned char** dstG = outimage->GetPixelArray(1);
    unsigned char** dstB = outimage->GetPixelArray(2);


    for(int y=0; y<maskimage->GetHeight(); y++)
    {
        for(int x=0; x<maskimage->GetWidth(); x++)
        {
            if(mpix[y][x] > 0)
            {
                dstR[y][x] = srcR[y][x];
                dstG[y][x] = srcG[y][x];
                dstB[y][x] = srcB[y][x];
            }
            else
            {
                dstR[y][x] = bgcolor.r;
                dstG[y][x] = bgcolor.g;
                dstB[y][x] = bgcolor.b;
            }
        }
    }

    return outimage;
}

    ObjectList label_Objects2(Image *imageP, Image **labelP, unsigned background);

// Return objects found in the image.
BinaryObjects* ExImage::GetObjects()
{
    Image* labelImage;
    ObjectList objlist;
    objlist = label_Objects2(m_image, &labelImage, 0);
```

```
	// Make sure objlist is not NULL
	if(!objlist)
		objlist = new_LL();
	BinaryObjects *binobjs = new BinaryObjects(objlist, labelImage);
	return binobjs;
}

int ExImage::FindPeakValue(int band)
{
	// Find peak.
	unsigned char **pix = (unsigned char**)m_image->image_ptr[band]->rptr;
	long* histogram = new long[256];

	for(int i=0; i<256; i++)
		histogram[i] = 0;

	for(int y=0; y<GetHeight(); y++)
	{
		for(int x=0; x<GetWidth(); x++)
		{
			histogram[pix[y][x]]++;
		}
	}

	// Find the peak.
	int high = 0;
	for(i=0; i<256; i++)
	{
		if(histogram[i] > histogram[high])
			high = i;
	}

	delete [] histogram;

	return (high);

}

ColorType ExImage::GetAverageColor()
{
	long rsum = 0, gsum = 0, bsum = 0;

	unsigned char** pixR = (unsigned char**)m_image->image_ptr[0]->rptr;
	unsigned char** pixG = (unsigned char**)m_image->image_ptr[1]->rptr;
	unsigned char** pixB = (unsigned char**)m_image->image_ptr[2]->rptr;

	for(int y=0; y<getNoOfRows_Image(m_image); y++)
		for(int x=0; x<getNoOfCols_Image(m_image); x++)
		{
			rsum += pixR[y][x];
			gsum += pixG[y][x];
			bsum += pixB[y][x];
		}

	ColorType color;
	color.r = rsum/(getNoOfRows_Image(m_image)*getNoOfCols_Image(m_image));
	color.g = gsum/(getNoOfRows_Image(m_image)*getNoOfCols_Image(m_image));
```

```
	color.b = bsum/(getNoOfRows_Image(m_image)*getNoOfCols_Image(m_image));

	return color;
}

// Dump properties of all objects.
void BinaryObjects::Dump()
{
	head_LL(m_objectlist);  // set linked list pointer to the head

	for(int i=0; i<size_LL(m_objectlist); i++)
	{
		Object *obj = ((Object *)retrieve_LL(m_objectlist));
		CString msg;
		msg.Format("label=%d; R=%d G=%d B=%d; xmin=%d ymin=%d xmax=%d ymax=%d;\n
eigenratio=%f; orientation=%f; horizontal cog=%f vertical cog=%f; area=%f",
			obj->label, obj->pixel.r, obj->pixel.g, obj->pixel.b, obj->x_min, obj-
>y_min, obj->x_max, obj->y_max, obj->prop.eig_ratio, obj->prop.orientation, obj-
>prop.h_cog, obj->prop.v_cog, obj->prop.area);
		AfxMessageBox(msg);
		next_LL(m_objectlist);
	}
}

int _cdecl MatchInt(void* content, void* lookforP)
{
	return(*((int*)content) == *((int*)lookforP));
}

// Remove objects whose sizes are equal to or less than minarea.
void BinaryObjects::FilterOut(int minarea)
{
	if(m_objectlist->listlength == 0)
		return;

	getProp_Objects(m_objectlist, m_labelImage);

	head_LL(m_objectlist);
	previous_LL(m_objectlist);

	//int** pix = (int**)m_labelImage->image_ptr[0]->rptr;

	for(;;)
	{
		Object* obj = ((Object *)retrieveNext_LL(m_objectlist));

		if(obj->prop.area < minarea)
		{
			// Erase pixels belonging to this object from the label image
			for(int y=obj->y_min; y<=obj->y_max; y++)
				for(int x=obj->x_min; x<=obj->x_max; x++)
				{
					if(((int*)m_labelImage->image_ptr[0]->rptr[y])[x] == obj-
>label)
					{
						((int*)m_labelImage->image_ptr[0]->rptr[y])[x] = 0;
```

```
                        }
                    }

                removenext_LL(m_objectlist);
            }
            else
                next_LL(m_objectlist);

            if(istail_LL(m_objectlist))
                break;
        }
}

ExImage* BinaryObjects::CreateContourImage()
{
        Image* returnimg;
        returnimg = new_Image(PBM, BINARY, 1, getNoOfRows_Image(m_labelImage),
            getNoOfCols_Image(m_labelImage), CVIP_BYTE, REAL);
        /*
        returnimg = new_Image(TIF, BINARY,
      1, getNoOfRows_Image(m_labelImage), getNoOfCols_Image(m_labelImage),
         CVIP_BYTE, REAL);
        */
    // Extract chain code from each object, and draw them onto returnimg.

        // dump labelled image

        CString msg;
        /*
        msg.Format("#    bands=%d    width=%d    height=%d    format=%d    type=%d,    space=%d",
m_labelImage->bands,
            getNoOfCols_Image(m_labelImage),             getNoOfRows_Image(m_labelImage),
m_labelImage->image_format,
            m_labelImage->image_ptr[0]->data_type, m_labelImage->color_space);
        AfxMessageBox(msg);
        msg.Format("# bands=%d width=%d height=%d format=%d type=%d, space=%d", returnimg-
>bands,
            getNoOfCols_Image(returnimg),     getNoOfRows_Image(returnimg),     returnimg-
>image_format,
            returnimg->image_ptr[0]->data_type, returnimg->color_space);
        AfxMessageBox(msg);
        */

    head_LL(m_objectlist);

    for(;;)
    {
            int ray_x;

        Object *obj = ((Object*)retrieve_LL(m_objectlist));
            //shootRay(m_labelImage, obj->label, &ray_x, &ray_y, obj->x_min, obj->y_min,
obj->x_max, obj->y_max);
            for(int c=obj->x_min; c<=obj->x_max; c++)
            {
                if(((int*)m_labelImage->image_ptr[0]->rptr[obj->y_min])[c]    ==    obj-
>label)
                {
```

```
                    ray_x = c;
                    break;
                }
            }
        ChainCode* cc = new_ChainCode(obj->y_min, ray_x, obj->label);
        if(build_LineChainCode(cc, m_labelImage, obj->x_min, obj->y_min, obj->x_max, obj->y_max)==0)
                AfxMessageBox("Error building chain code!");
            draw_ChainCode(cc, returnimg);
        if(istail_LL(m_objectlist))
            break;
        next_LL(m_objectlist);
    }

    ExImage* newimg = new ExImage;
    newimg->m_image = returnimg;

    return newimg;
}

ExImage* BinaryObjects::CreateImage()

    Image* returnimg;
    returnimg = new_Image(PGM, GRAY_SCALE, 1, getNoOfRows_Image(m_labelImage),
        getNoOfCols_Image(m_labelImage), CVIP_BYTE, REAL);

    for(int y=0; y<getNoOfRows_Image(m_labelImage); y++)
        for(int x=0; x<getNoOfCols_Image(m_labelImage); x++)
        {
            if(((int*)m_labelImage->image_ptr[0]->rptr[y])[x] > 0)
                returnimg->image_ptr[0]->rptr[y][x] = 255U;
            else
                returnimg->image_ptr[0]->rptr[y][x] = 0;
        }

    ExImage* newimg = new ExImage;
    newimg->m_image = returnimg;

    return newimg;
}

ConnectivityGraph<Edge *>* BinaryObjects::CreateJunctionGraph(ObjectInfo* objinfo, std::vector<Junction*>& junclist /* output */)
{
    // Find all junctions.
    junclist = ComputeJunctions(objinfo);

    TRACE("There are %d pre-junctions\n", junclist.size());

    ConnectivityGraph<Edge *>* cg = new ConnectivityGraph<Edge *>(false, junclist.size());

    //int *degreelist = new int[junclist.size()];

    // Copy degrees.
    //for(int i=0; junclist.size(); i++)
    //{
```

```
//    degreelist[i] = junclist[i]->degree;
//}

// Connect nodes.
for(int i=0; i<junclist.size(); i++)
{
      int c = 0;
      float oldlen;
      while(junclist[i]->m_neighborPixels.size() != 0)
      {
            Edge *edge = new Edge;
            TRACE("j[%d]'s   next   neighbor   is   %d\n",   i,   *(junclist[i]-
>m_neighborPixels.begin()));
            int    nextjunc    =    FindNextJunction(junclist,    *(junclist[i]-
>m_neighborPixels.begin()), i, *edge);
            // If nextjunc is i itself, then this is a terminal loop.
            // Simply ignore it.
            if(nextjunc != i)
            {
                  TRACE("Inserting edge (%d, %d)\n", i, nextjunc);

                  //  If  there  is  already  an  edge  (i,  nextjunc),  ignore  the  new
edge.
                  if(!cg->IsEdge(i, nextjunc))
                  {
                        oldlen = edge->m_length;
                        cg->InsertEdge(i, nextjunc, edge);
                  }
                  else if(edge->m_length < oldlen)
                  {
                        cg->DeleteEdge(i, nextjunc);
                        cg->InsertEdge(i, nextjunc, edge);
                  }

                  // Erase the neighbor that leads to nextjunc and
                  // erase the neighbor that leads to i so as to avoid
                  // duplicate processing of the same edge.
                  TRACE("Erasing neighbor %d from j[%d] and %d from j[%d]\n", edge-
>GetNeighbor(i), i, edge->GetNeighbor(nextjunc), nextjunc);
                  junclist[i]->m_neighborPixels.erase(edge->GetNeighbor(i));
                  junclist[nextjunc]->m_neighborPixels.erase(edge-
>GetNeighbor(nextjunc));
                  //degreelist[i]--;
                  c++;
            }
            else
            {
                  // Erase the neighbor pixel that leads to the loop
                  //  In  case  of  loop,  edge->GetNeighbor(0)  returns  the  first
neighbor
                  // and edge->GetNeighbor(1) returns the last neighbor
                  junclist[i]->m_neighborPixels.erase(edge->GetNeighbor(0));
                  junclist[i]->m_neighborPixels.erase(edge->GetNeighbor(1));
            }
      }
}
```

```
	TRACE("There are %d post edges\n", cg->GetNumEdges());

	return cg;
}

int BinaryObjects::FindNextJunction(std::vector<Junction *> junclist, int neighbor, int
start, Edge& edge)
{
	// FOR DEBUG
	FILE* out = fopen("junction.txt", "w");

	int** pix = (int**)m_labelImage->image_ptr[0]->rptr;

	int deg;

	int x = junclist[start]->m_x;
	int y = junclist[start]->m_y;

	edge.m_junc1 = start;

	// Keep track of all coordinates.
	edge.m_xarray.clear();
	edge.m_yarray.clear();

	edge.m_xarray.push_back(x);
	edge.m_yarray.push_back(y);

	edge.m_xmin = 1000000;
	edge.m_xmax = -1;
	edge.m_ymin = 1000000;
	edge.m_ymax = -1;

	// FOR DEBUG
	fprintf(out, "finding the next junction for junction %d\n", start);

	do
	{
		fprintf(out, "neighbor=%d\n", neighbor);
		switch(neighbor)
		{
		case 0:
			x = x+1;
			y = y;
			edge.m_length += 1.0f;
			break;
		case 1:
			x = x+1;
			y = y+1;
			edge.m_length += 1.41421356f;
			break;
		case 2:
			x = x;
			y = y+1;
			edge.m_length += 1.0f;
			break;
		case 3:
			x = x-1;
```

```
        y = y+1;
        edge.m_length += 1.41421356f;
        break;
case 4:
        x = x-1;
        y = y;
        edge.m_length += 1.0f;
        break;
case 5:
        x = x-1;
        y = y-1;
        edge.m_length += 1.41421356f;
        break;
case 6:
        x = x;
        y = y-1;
        edge.m_length += 1.0f;
        break;
case 7:
        x = x+1;
        y = y-1;
        edge.m_length += 1.41421356f;
        break;
}

if(x < edge.m_xmin)
        edge.m_xmin = x;
if(x > edge.m_xmax)
        edge.m_xmax = x;
if(y < edge.m_ymin)
        edge.m_ymin = y;
if(y > edge.m_ymax)
        edge.m_ymax = y;

edge.m_xarray.push_back(x);
edge.m_yarray.push_back(y);

// Is the current position a junction?

// neighbor
//  5 6 7
//  4   0
//  3 2 1

int nextNeighbor;

deg=0;

if((pix[y][x+1]  != 0)&&(neighbor != 4))
{
        deg++;
        nextNeighbor = 0;
}
if((pix[y+1][x]  != 0)&&(neighbor != 6))
{
        deg++;
        nextNeighbor = 2;
```

```
        }
        if((pix[y][x-1] != 0)&&(neighbor != 0))
        {
              deg++;
              nextNeighbor = 4;
        }
        if((pix[y-1][x] != 0)&&(neighbor != 2))
        {
              deg++;
              nextNeighbor = 6;
        }
        if((pix[y+1][x+1] != 0)&&(pix[y][x+1]==0)&&(pix[y+1][x]==0)&&(neighbor != 5))
        {
              deg++;
              nextNeighbor = 1;
        }
        if((pix[y+1][x-1] != 0)&&(pix[y+1][x]==0)&&(pix[y][x-1]==0)&&(neighbor != 7))
        {
              deg++;
              nextNeighbor = 3;
        }
        if((pix[y-1][x-1] != 0)&&(pix[y-1][x]==0)&&(pix[y][x-1]==0)&&(neighbor != 1))
        {
              deg++;
              nextNeighbor = 5;
        }
        if((pix[y-1][x+1] != 0)&&(pix[y-1][x]==0)&&(pix[y][x+1]==0)&&(neighbor != 3))
        {
              deg++;
              nextNeighbor = 7;
        }

        fprintf(out, "degree=%d nextneighbor=%d\n", deg, nextNeighbor);
        neighbor = nextNeighbor;
  } while(deg == 1);

  // Compute the angle the edge forms at each junction (with respect to x-axis).
  // 1. angle at junction 1
  int chainlen = edge.m_xarray.size();

  int dx = edge.m_xarray[min(4, chainlen-1)] - edge.m_xarray[0];
  int dy = edge.m_yarray[min(4, chainlen-1)] - edge.m_yarray[0];

  if(dx == 0)
  {
        if(dy < 0)
              edge.m_angle1 = HPI;
        else
              edge.m_angle1 = -HPI;
  }
  else
        edge.m_angle1 = atan2((double)-dy, (double)dx);

  dx = edge.m_xarray[max(chainlen-5, 0)] - edge.m_xarray[chainlen-1];
  dy = edge.m_yarray[max(chainlen-5, 0)] - edge.m_yarray[chainlen-1];

  if(dx == 0)
```

```
	{
		if(dy < 0)
			edge.m_angle2 = HPI;
		else
			edge.m_angle2 = -HPI;
	}
	else
		edge.m_angle2 = atan2((double)-dy, (double)dx);

	// Return the ID of the junction
	for(int i=0; i<junclist.size(); i++)
	{
		if((junclist[i]->m_x == x)&&(junclist[i]->m_y == y))
		{
			edge.m_junc2 = i;
			fclose(out);
			return i;
		}
	}

	for(i=0; i<junclist.size(); i++)
	{
		fprintf(out, "junc %d=%d %d\n", i, junclist[i]->m_x, junclist[i]->m_y);
	}
	fclose(out);
	ASSERT(FALSE);
	return -1;  // Error!!!
}

ConnectivityGraph<float>*  BinaryObjects::CreateEdgeGraph(ConnectivityGraph<Edge  *>*  jg,
ObjectInfo* objinfo)
{
	// Edge Graph
	ConnectivityGraph<float>*   eg   =   new   ConnectivityGraph<float>(false,   jg-
>GetNumEdges());

	//CString msg;
	//msg.Format("junction graph: number of edges=%d", jg->GetNumEdges());
	//AfxMessageBox(msg);

	// For each edge-to-edge connection, compute the angle between the edges.

	int JGsize = jg->GetSize();

	for(int i=0; i<JGsize; i++)
	{
		for(int j=i+1; j<JGsize; j++)
		{
			if(jg->IsEdge(i, j))
			{
				for(int k=0; k<JGsize; k++)
				{
					if(jg->IsEdge(j, k) && (i != k))
					{
						// Compute the angle between i,j,k

						//      i
```

```
                                        //      ^
                                        //     / \
                                        //    /   \
                                        //   /     \
                                        //  j       k

                                        if(!eg->IsEdge(jg->GetEdgeID(i,  j),  jg->GetEdgeID(j,
k)))
                                        {
                                              float  angle  =  jg->GetEdge(i,j)->GetAngle(j)  -
jg->GetEdge(j,k)->GetAngle(j);
                                              if(angle < 0.0f)
                                                    angle += DPI;
                                              if(angle > PI)
                                                    angle = DPI - angle;
                                              eg->InsertEdge(jg->GetEdgeID(i,     j),     jg-
>GetEdgeID(j,k), angle);
                                              //CString msg;
                                              //msg.Format("Inserting  edge  %d  %d  for  %d  %d
%d", jg->GetEdgeID(i,j), jg->GetEdgeID(j,k), i, j, k);
                                              //AfxMessageBox(msg);
                                        }
                                  }
                                  else if(jg->IsEdge(i, k) && (j != k))
                                  {
                                        // Compute the angle between i,j,k

                                        //      j
                                        //      ^
                                        //     / \
                                        //    /   \
                                        //   /     \
                                        //  i       k

                                        if(!eg->IsEdge(jg->GetEdgeID(i,  j),  jg->GetEdgeID(i,
k)))
                                        {
                                              float  angle  =  jg->GetEdge(i,j)->GetAngle(i)  -
jg->GetEdge(i,k)->GetAngle(i);
                                              if(angle < 0.0f)
                                                    angle += DPI;
                                              if(angle > PI)
                                                    angle = DPI - angle;
                                              eg->InsertEdge(jg->GetEdgeID(i,     j),     jg-
>GetEdgeID(i, k), angle);
                                              CString msg;
                                              msg.Format("Inserting edge %d %d for %d %d %d",
jg->GetEdgeID(i,j), jg->GetEdgeID(i,k), i, j, k);
                                              AfxMessageBox(msg);
                                        }
                                  }

                            }
                      }
                }
          }
          return eg;
```

```
}

ExImage* BinaryObjects::CreateJunctionImage(std::vector<ObjectInfo*> objinfo)
{
	Image* returnimg;
	returnimg = new_Image(PPM, RGB, 3, getNoOfRows_Image(m_labelImage),
		getNoOfCols_Image(m_labelImage), CVIP_BYTE, REAL);

	head_LL(m_objectlist);

	int** srcpix = (int**)m_labelImage->image_ptr[0]->rptr;
	unsigned char** destpixR = (unsigned char**)returnimg->image_ptr[0]->rptr;
	unsigned char** destpixG = (unsigned char**)returnimg->image_ptr[1]->rptr;
	unsigned char** destpixB = (unsigned char**)returnimg->image_ptr[2]->rptr;

	for(int i=0; i<size_LL(m_objectlist); i++)
	{
		Object* obj = (Object*)retrieve_LL(m_objectlist);
		for(int y = obj->y_min; y <= obj->y_max; y++)
		{
			for(int x = obj->x_min; x <= obj->x_max; x++)
			{
				if(srcpix[y][x] == obj->label)
				{
					destpixR[y][x] = 255;
					destpixG[y][x] = 255;
					destpixB[y][x] = 255;
				}
			}
		}
		linked_list* jlist = objinfo[i]->junctions;
		head_LL(jlist);
		for(int j=0; j<size_LL(jlist); j++)
		{
			Junction* junc = (Junction*)retrieve_LL(jlist);
			for(int k=0; k<junc->m_degree; k++)
			{
				int jx = junc->m_x, jy = junc->m_y;
				switch(junc->m_degree)
				{
				case 3:
					destpixR[jy][jx] = 255;
					destpixG[jy][jx] = 0;
					destpixB[jy][jx] = 0;
					break;
				case 4:
					destpixR[jy][jx] = 255;
					destpixG[jy][jx] = 255;
					destpixB[jy][jx] = 0;
					break;
				case 5:
					destpixR[jy][jx] = 0;
					destpixG[jy][jx] = 255;
					destpixB[jy][jx] = 0;
					break;
				case 6:
					destpixR[jy][jx] = 0;
```

```
                    destpixG[jy][jx] = 255;
                    destpixB[jy][jx] = 255;
                    break;
                case 7:
                    destpixR[jy][jx] = 0;
                    destpixG[jy][jx] = 0;
                    destpixB[jy][jx] = 255;
                    break;
                }
            }
            next_LL(jlist);
        }

        next_LL(m_objectlist);
    }

    ExImage *newimg = new ExImage;
    newimg->m_image = returnimg;

    return newimg;
}

void ObjectInfo::Dump()
{
    CString msg;
    msg.Format("xmin=%d  xmax=%d  ymin=%d  ymax=%d\nperimeter=%d  area=%d  eigenratio=%f
orientation=%f\nxcenter=%f ycenter=%f numjunc=%d",
        xmin, xmax, ymin, ymax, perimeter, area, eigenratio, orientation, xcenter,
ycenter, numjunc);
    AfxMessageBox(msg);
}

std::vector<ObjectInfo*> BinaryObjects::GetObjectInfo(bool buildChaincode)
{
    std::vector<ObjectInfo*> infolist;

    if(m_objectlist->listlength == 0)
        return infolist;

    getProp_Objects(m_objectlist, m_labelImage);

    head_LL(m_objectlist);

    for(int i=0; i<size_LL(m_objectlist); i++)
    {
        int ray_x;
        Object* obj = ((Object*)retrieve_LL(m_objectlist));
        //shootRay(m_labelImage, obj->label, &ray_x, &ray_y, obj->x_min, obj->y_min,
obj->x_max, obj->y_max);
        for(int c=obj->x_min; c<=obj->x_max; c++)
        {
            if(((int*)m_labelImage->image_ptr[0]->rptr[obj->y_min])[c]    ==    obj-
>label)
            {
                ray_x = c;
                break;
```

```
			}
		}

		/*
		for(int y=obj->y_min; y<=obj->y_max; y++)
		{
			for(int x=obj->x_min; x<=obj->x_max; x++)
			{
				CString msg;
				msg.Format("i=%d  objlabel=%d,  label=%d  xmin=%d  xmax=%d  ymin=%d
ymax=%d   xstart=%d   ystart=%d",   i,   obj->label,   ((int*)m_labelImage->image_ptr[0]-
>rptr[y])[x], obj->x_min, obj->x_max, obj->y_min, obj->y_max);
				AfxMessageBox(msg);
			}
		}
		*/

		ObjectInfo* info = new ObjectInfo;

		if(buildChaincode)
		{
			info->chain = new_ChainCode(obj->y_min, ray_x, obj->label);
			if(!build_LineChainCode(info->chain,   m_labelImage,   obj->x_min,   obj-
>y_min, obj->x_max, obj->y_max))
				AfxMessageBox("Error building chain code");
			//print_ChainCode(infolist[i]->chain, "chaininfo.txt");
			info->perimeter = info->chain->no_of_vectors;
		}
		info->area = obj->prop.area;
		info->label = obj->label;
		info->eigenratio = obj->prop.eig_ratio;
		info->orientation = obj->prop.orientation;
		info->xcenter = obj->prop.h_cog;
		info->ycenter = obj->prop.v_cog;
		info->xmin = obj->x_min;
		info->xmax = obj->x_max;
		info->ymin = obj->y_min;
		info->ymax = obj->y_max;
		//info->junctions = ComputeJunctions(info);
		infolist.push_back(info);
		next_LL(m_objectlist);
	}

	return infolist;
}

std::vector<ExImage*> BinaryObjects::CreateColorImages(std::vector<ObjectInfo*> infolist,
ImageIO* original)
{
	std::vector<ExImage*> imglist;

	int** lab = (int**)m_labelImage->image_ptr[0]->rptr;
	unsigned char** origR = original->GetPixelArray(0);
	unsigned char** origG = original->GetPixelArray(1);
	unsigned char** origB = original->GetPixelArray(2);

	for(int i=0; i<infolist.size(); i++)
```

```
    {
        ObjectInfo* info = infolist[i];
        // Create a color image for the object.
        Image* returnimg = new_Image(PPM, RGB, 3, info->ymax - info->ymin + 1,
            info->xmax - info->xmin + 1, CVIP_BYTE, REAL);

        unsigned char** destR = (unsigned char**)returnimg->image_ptr[0]->rptr;
        unsigned char** destG = (unsigned char**)returnimg->image_ptr[1]->rptr;
        unsigned char** destB = (unsigned char**)returnimg->image_ptr[2]->rptr;

        int count = 0;

        for(int y=info->ymin; y<=info->ymax; y++)
        {
            for(int x=info->xmin; x<=info->xmax; x++)
            {
                int ny = y - info->ymin;
                int nx = x - info->xmin;

                if(lab[y][x] == info->label)
                {
                    destR[ny][nx] = origR[y][x];
                    destG[ny][nx] = origG[y][x];
                    destB[ny][nx] = origB[y][x];
                    count++;
                }
                else
                {
                    destR[ny][nx] = (unsigned char)0;
                    destG[ny][nx] = (unsigned char)0;
                    destB[ny][nx] = (unsigned char)0;
                }
            }
        }
        ExImage* newimage = new ExImage;
        newimage->m_image = returnimg;
        imglist.push_back(newimage);
    }
    return imglist;
}

ExImage* BinaryObjects::CreateColorImage(ObjectInfo* info, ImageIO* original)
{

    int** lab = (int**)m_labelImage->image_ptr[0]->rptr;
    unsigned char** origR = original->GetPixelArray(0);
    unsigned char** origG = original->GetPixelArray(1);
    unsigned char** origB = original->GetPixelArray(2);

        // Create a color image for the object.
        Image* returnimg = new_Image(PPM, RGB, 3, info->ymax - info->ymin + 1,
            info->xmax - info->xmin + 1, CVIP_BYTE, REAL);

        unsigned char** destR = (unsigned char**)returnimg->image_ptr[0]->rptr;
        unsigned char** destG = (unsigned char**)returnimg->image_ptr[1]->rptr;
        unsigned char** destB = (unsigned char**)returnimg->image_ptr[2]->rptr;
```

```
		int count = 0;

		for(int y=info->ymin; y<=info->ymax; y++)
		{
			for(int x=info->xmin; x<=info->xmax; x++)
			{
				int ny = y - info->ymin;
				int nx = x - info->xmin;

				if(lab[y][x] == info->label)
				{
					destR[ny][nx] = origR[y][x];
					destG[ny][nx] = origG[y][x];
					destB[ny][nx] = origB[y][x];
					count++;
				}
				else
				{
					destR[ny][nx] = (unsigned char)0;
					destG[ny][nx] = (unsigned char)0;
					destB[ny][nx] = (unsigned char)0;
				}
			}
		}
		ExImage* newimage = new ExImage;
		newimage->m_image = returnimg;

		return newimage;
}

std::vector<Junction*> BinaryObjects::ComputeJunctions(ObjectInfo* objinfo)
{
	std::vector<Junction*> junclist;

	int** pix = (int**)m_labelImage->image_ptr[0]->rptr;
	int label = objinfo->label;

	for(int y=objinfo->ymin; y<=objinfo->ymax; y++)
		for(int x=objinfo->xmin; x<=objinfo->xmax; x++)
		{
			std::set<int> neighbor;

			if(pix[y][x] == label)
			{
				// neighbor
				//  5 6 7
				//  4   0
				//  3 2 1

				int deg=0;
				if((x!=objinfo->xmax) && (pix[y][x+1] == label))
				{
					deg++;
					neighbor.insert(0);
				}
				if((y!=objinfo->ymin) && (pix[y-1][x] == label))
				{
```

```
				deg++;
				neighbor.insert(6);
			}
			if((x!=objinfo->xmin) && (pix[y][x-1] == label))
			{
				deg++;
				neighbor.insert(4);
			}
			if((y!=objinfo->ymax) && (pix[y+1][x] == label))
			{
				deg++;
				neighbor.insert(2);
			}
			if((y!=objinfo->ymin) && (x!=objinfo->xmax) && (pix[y-1][x+1] ==
label))
			{
				if((neighbor.find(6)==neighbor.end())                &&
(neighbor.find(0)==neighbor.end()))
				{
					deg++;
					neighbor.insert(7);
				}
			}
			if((y!=objinfo->ymin) && (x!=objinfo->xmin) && (pix[y-1][x-1] ==
label))
			{
				if((neighbor.find(4)==neighbor.end())                &&
(neighbor.find(6)==neighbor.end()))
				{
					deg++;
					neighbor.insert(5);
				}
			}
			if((y!=objinfo->ymax) && (x!=objinfo->xmin) && (pix[y+1][x-1] ==
label))
			{
				if((neighbor.find(2)==neighbor.end())                &&
(neighbor.find(4)==neighbor.end()))
				{
					deg++;
					neighbor.insert(3);
				}
			}
			if((y!=objinfo->ymax) && (x!=objinfo->xmax) && (pix[y+1][x+1] ==
label))
			{
				if((neighbor.find(0)==neighbor.end())                &&
(neighbor.find(2)==neighbor.end()))
				{
					deg++;
					neighbor.insert(1);
				}
			}

			// Modify neighborhood list based on the following rules:
			// if the neighbor is odd and there is an adjacent neighbor,
			// remove the neighbor from the list.
```

```
                        /*
                        std::set<int>::iterator it = neighbor.begin();
                        int c = 0;
                        while(it != neighbor.end())
                        {
                              if(*it % 2 == 1)
                              {
                                    int n = *it;
                                    int n1 = (n != 0) ? n-1 : 7;
                                    int n2 = (n != 7) ? n+1 : 0;
                                    if((neighbor.find(n1)     !=     neighbor.end())     ||
(neighbor.find(n2) != neighbor.end()))
                                    {
                                          it = neighbor.erase(it);
                                          deg--;
                                    }
                              }
                              else
                                    it++;
                              if(c>10)
                                    AfxMessageBox("ERROR");
                              c++;
                        }
*/
                        if(deg != 2)
                        {
                              Junction *junc = new Junction;
                              junc->m_degree = deg;
                              junc->m_x = x;
                              junc->m_y = y;
                              junc->m_neighborPixels = neighbor;
                              junclist.push_back(junc);
                        }
                  }
            }
      return junclist;
}


void    BinaryObjects::MergeJunctions(ConnectivityGraph<Edge     *>     *jgraph,     float
mergeDistance)
{
      // juncs keeps track of junctions that belong to "short" edges.
      std::set<int> juncs;
      std::set<int>::iterator it;

      Point* juncpos = new Point[jgraph->GetSize()];

      for(int i=0; i<jgraph->GetSize(); i++)
      {
            // Find edges that are short enough and put their junctions into juncs.
            // Remove those edges.
            for(int j=i+1; j<jgraph->GetSize(); j++)
            {
                  if(jgraph->IsEdge(i, j))
                  {
```

```
                    if(jgraph->GetEdge(i, j)->m_length < mergeDistance)
                    {
                        juncpos[i] = jgraph->GetEdge(i, j)->GetEndPosition(i);
                        juncs.insert(i);
                        juncs.insert(j);
                        jgraph->DeleteEdge(i, j);
                    }
                }
            }
        }

        std::vector<std::vector<int> > groups;

        // Look at junctions' connectivity and divide them into groups
        // of connected junctions.
        while(!juncs.empty())
        {
            std::vector<int> list;
            groups.push_back(list);

            int current = *juncs.begin();
            groups.rbegin()->push_back(current);
            juncs.erase(juncs.begin());

            std::vector<int> worklist;
            do
            {
                it = juncs.begin();
                while(it!=juncs.end())
                {
                    if(jgraph->IsEdge(current, *it))
                    {
                        worklist.push_back(*it);
                        it = juncs.erase(it);
                    }
                    else
                        it++;
                }
                if(worklist.empty())
                    break;
                current = *worklist.begin();
                worklist.erase(worklist.begin());
                groups.rbegin()->push_back(current);
            } while(!worklist.empty());
        }

        int* newjuncIndex = new int[groups.size()];
        Point* newjuncPos = new Point[groups.size()];

        // Compute the centroid for each group and make it a new junction.
        for(i=0; i<groups.size(); i++)
        {
            float sumx = 0.0f;
            float sumy = 0.0f;
            for(int j=0; j<groups[i].size(); j++)
            {
                sumx += juncpos[groups[i][j]].x;
```

```
				sumy += juncpos[groups[i][j]].y;
			}
			sumx /= groups[i].size();
			sumy /= groups[i].size();
			Point pt(ROUND(sumx), ROUND(sumy));
			newjuncPos[i] = pt;
			newjuncIndex[i] = groups[i][0];
		}

		for(i=0; i<groups.size(); i++)
		{
			for(int j=0; j<groups[i].size(); j++)
			{
				// Modify all edges that are connected to groups[i][j]
				for(int k=0; k<jgraph->GetSize(); k++)
				{
					if(jgraph->IsEdge(groups[i][j], k))
					{
						Edge* edge = jgraph->GetEdge(groups[i][j], k);
						//AddSegmentEnd(edge,  newjuncPos[i],  newjuncIndex[i],  k,
mergeDistance);

						// Compute the angle the edge forms at each junction (with
respect to x-axis).
						// 1. angle at junction 1
						int chainlen = edge->m_xarray.size();

						int endIndex;

						if(newjuncIndex[i] == edge->m_junc1)
							endIndex = 0;
						else
							endIndex = chainlen - 1;

						int dx = newjuncPos[i].x - edge->m_xarray[endIndex];
						int dy = newjuncPos[i].y - edge->m_yarray[endIndex];

						if(dx == 0)
						{
							if(dy > 0)
							{
								if(newjuncIndex[i] == edge->m_junc1)
									edge->m_angle1 = HPI;
								else
									edge->m_angle2 = HPI;
							}
							else
							{
								if(newjuncIndex[i] == edge->m_junc1)
									edge->m_angle1 = -HPI;
								else
									edge->m_angle2 = -HPI;
							}
						}
						else
						{
							if(newjuncIndex[i] == edge->m_junc1)
```

```
                                        edge->m_angle1        =        atan2((double)-dy,
(double)dx);
                                else
                                        edge->m_angle2        =        atan2((double)-dy,
(double)dx);
                        }
                    }
                }
            }
        }
}

void BinaryObjects::AddSegmentEnd(Edge* edge, Point newPos, int newIndex, int otherIndex,
float mergeDistance)
{
        // Connect the edge to the newly formed junction.

        ASSERT(edge->m_junc1 == otherIndex || edge->m_junc2 == otherIndex);
        if(edge->m_junc1 == otherIndex)
                edge->m_junc2 = newIndex;
        else
                edge->m_junc1 = newIndex;
}

/*
// Helper function for GetObjectInfo.
void  BinaryObjects::ComputeJunctions(ObjectInfo*  objinfo,  Junction  **junctions,  int
&numjunc)
{
        int xmin = objinfo->xmin;
        int ymin = objinfo->ymin;
        int xmax = objinfo->xmax;
        int ymax = objinfo->ymax;

        // Convert the chain code into a list of (x,y).
        int *xP=NULL, *yP=NULL;
        getXY_ChainCode(objinfo->chain, &xP, &yP);
        // count is a counter for each coordinate in the chain code.
        int** count = new int*[ymax-ymin+1];
        for(int i=0; i<ymax-ymin+1; i++)
        {
                count[i] = new int[xmax-xmin+1];
                // Reset counter.
                for(int j=0; j<xmax-xmin+1; j++)
                        count[i][j] = 0;
        }

        numjunc = 0;

        for(i=0; i<objinfo->chain->no_of_vectors; i++)
        {
                // Increment numjunc if the junction degree is greater than 2.
                if(count[yP[i]-ymin][xP[i]-xmin] == 2)
                {
                        numjunc++;
```

```
		}
		(count[yP[i]-ymin][xP[i]-xmin])++;
	}

	// No junction.
	if(numjunc == 0)
	{
		*junctions = NULL;
		return;
	}

	Junction* newjunctions = new Junction[numjunc];

	int jc = 0;
	for(i=0; i<ymax-ymin+1; i++)
		for(int j=0; j<xmax-xmin+1; j++)
		{
			if(count[i][j] >= 3)
			{
				newjunctions[jc].degree = count[i][j];
				newjunctions[jc].x = j+xmin;
				newjunctions[jc].y = i+ymin;
				jc++;
			}
		}
	*junctions = newjunctions;
	ASSERT(jc == numjunc);

	// Clean up.
	delete [] xP;
	delete [] yP;
*/

typedef HashTable *ObjectHash;

#define HASH_SIZE 251U

#define hash_Object(label) (((unsigned)(label))%HASH_SIZE)

static void addto_Objects(ObjectHash hashP, int next_label, Color pixel, int y_pos, int
x_pos);
static void update_Objects(ObjectHash hashP, int object_label, int y_pos, int x_pos);
static void combine_Objects(ObjectHash hashP, int b, int c, int *xmin, int *xmax, int
*ymin, int *ymax);
static ObjectList hash2List_Objects(ObjectHash hashP);

static void makegraymap_Objects(ColorHistogram *chP);
static Matrix *color2index_Image(ColorHistogram *chP, Image *imageP );

/*
 * label recycling routines
 */
static void initLabelStack(void);
static int getNextLabel(void);
static void recycleLabel(int label);
```

```
/*
 * global variables used by label recycling routines
 */
static int label_count;
static Stack label_stackP;

ObjectList
label_Objects2(
                     Image *imageP,
                     Image **labelP,
                     unsigned background
                     )
{
      register int x, y=0, i, j;
      int A, B, C, D, rows, cols;
      byte *pixarray2, *pixarray1; /* pixel arrays */
      int *labarray1, *labarray2, *rowP; /* label arrays */
      int xmin, xmax, ymin, ymax, next_label;
      unsigned *A_LABEL, B_LABEL, C_LABEL, D_LABEL, MAX_LABEL;
      ObjectList listP;
      ObjectHash hashP;
      Object *objP;
      ColorHistogram *chP;
      ColorHistObject *mapP;
      Matrix *matrixP;
      ROI *roiP;
      const char *fn = "label";

      chP = new_ColorHist();

      if(getNoOfBands_Image(imageP) > 1) {
            compute_ColorHist(chP, imageP, 256);
            matrixP = color2index_Image(chP, imageP);

      }
      else {
            makegraymap_Objects(chP);
            matrixP = getBand_Image(imageP,0);
      }

      mapP = chP->histogram;
      if(mapP==NULL) return NULL;

      rows = getNoOfRows_Image(imageP);
      cols = getNoOfCols_Image(imageP);

      *labelP = new_Image(PGM, GRAY_SCALE, 1, rows, cols, CVIP_INTEGER, REAL);

      pixarray2 = (unsigned char*)getRow_Matrix(matrixP, 0);
      labarray2 = (int *)getRow_Image(*labelP, 0, 0);

      initLabelStack();
      hashP = new_HT(HASH_SIZE);

      /*
      * handle special case of first row
      */
```

```
for(x=0; x < cols; x++)
	if( pixarray2[x] != background ) {
		if( (x==0) || (pixarray2[x-1] != pixarray2[x]) ) {
			next_label = getNextLabel();
			addto_Objects(hashP, next_label, mapP[pixarray2[x]].pixel, y, x);
			labarray2[x] = next_label;
		}
		else {
			update_Objects(hashP, labarray2[x-1], y, x);
			labarray2[x] = labarray2[x-1];
		}
	}

	for(y=0; y < rows-1; y++)
	{

		pixarray1 = pixarray2;
		pixarray2 = (unsigned char*)getRow_Matrix(matrixP, y+1);

		labarray1 = labarray2;
		labarray2 = (int *)getRow_Image(*labelP, y+1, 0);

		for(x=0; x < cols-1; x++)
		{
			A = pixarray2[x+1];
			B = pixarray2[x];
			C = pixarray1[x+1];
			D = pixarray1[x];

			A_LABEL = (unsigned *) &labarray2[x+1];
			B_LABEL = labarray2[x];
			C_LABEL = labarray1[x+1];
			D_LABEL = labarray1[x];

			if (x == 0)
			{
				if (B != background)
					if(D == B)
					{
						B_LABEL = labarray2[x] = D_LABEL;
						update_Objects(hashP, B_LABEL, y+1, x);
					}
					else
					{
						next_label = getNextLabel();
						addto_Objects(hashP,                  next_label,
mapP[B].pixel,y+1,x);

						B_LABEL = labarray2[x] = next_label;
					}
			}  // x==0
			if (A != background)
			{
				if(D != A)
				{
					if(B != A)
					{
```

```
                        if(C != A)
                        {
                              next_label = getNextLabel();
                              addto_Objects(hashP,
next_label,mapP[A].pixel, y+1, x+1);
                              *A_LABEL = next_label;
                        } // C!=A
                        else
                        {
                              *A_LABEL = C_LABEL;
                              update_Objects(hashP,    *A_LABEL,    y+1,
x+1);
                        } // C==A
                  } // B!=A
                  else
                  {
                        if (C == A)
                        {
                              if (B_LABEL == C_LABEL)
                              {
                                    *A_LABEL = B_LABEL;
                              }
                              else
                              {
                                    combine_Objects(hashP,      B_LABEL,
C_LABEL, &xmin, &xmax,
                                          &ymin, &ymax);

                                    MAX_LABEL = MAX(B_LABEL,C_LABEL);
                                    *A_LABEL =  MIN(B_LABEL,C_LABEL);

                                    roiP = new_ROI();
                                    asgnImage_ROI(roiP,  *labelP,  xmin,
ymin, xmax-xmin+1,
                                          ymax-ymin+1);

                                    for(i=0; i < getNoOfRows_ROI(roiP);
i++) {
                                          rowP      =      (int      *)
getRow_ROI(roiP, i, 0);
                                          for(j=0;         j          <
getNoOfCols_ROI(roiP); j++, rowP++)
                                                if(*rowP == MAX_LABEL)
                                                      *rowP = *A_LABEL;
                                    }

                                    delete_ROI(roiP);
                              } // B_LABEL!=C_LABEL
                        } // C==A
                        else
                              *A_LABEL = B_LABEL;

                        update_Objects(hashP, *A_LABEL, y+1, x+1);
                  } // B!=A
            }  // D!=A
            else
            {
```

```
                    *A_LABEL = D_LABEL;
                    update_Objects(hashP, *A_LABEL, y+1, x+1);
                } // D==A
            } // A!=background
            else  // addition by Yusaku Sako. 7/26/99
            {
                if(B != background)
                    if((B == C) && (D != B) && (B_LABEL != C_LABEL))
                    {
                        combine_Objects(hashP, B_LABEL, C_LABEL, &xmin,
&xmax,

                            &ymin, &ymax);

                        MAX_LABEL = MAX(B_LABEL,C_LABEL);
                        B_LABEL = labarray2[x] = MIN(B_LABEL,C_LABEL);

                        roiP = new_ROI();
                        asgnImage_ROI(roiP, *labelP, xmin, ymin, xmax-
xmin+1,

                            ymax-ymin+1);

                        for(i=0; i < getNoOfRows_ROI(roiP); i++) {
                            rowP = (int *) getRow_ROI(roiP, i, 0);
                            for(j=0; j < getNoOfCols_ROI(roiP); j++,
rowP++)

                                if(*rowP == MAX_LABEL)
                                    *rowP = B_LABEL;
                        }

                        delete_ROI(roiP);
                        update_Objects(hashP, B_LABEL, y+1, x); //
erase

                    }
            }  // A==background
        } // for x
    } // for y

    if(hashP->table[hashP->key])
        listP = hash2List_Objects(hashP);
    else
        listP = NULL;

    /*
    fprintf(stderr, "Displaying Object Statistics...\n\n");
    fprintf(stderr, "Number of objects found = %u.\n\n", size_LL(listP));
    print_Objects(listP);
    fprintf(stderr, "\n\n              Press the <ENTER> key to continue");
    getchar();
    */

    // Bug fix by YS (3/24/00).
    // Don't wanna delete matrixP when number of bands is 1
    if(getNoOfBands_Image(imageP) > 1) {
        delete_Matrix(matrixP);
    }

    return listP;
```

```
}

static ObjectList hash2List_Objects(ObjectHash hashP)
{
   register int i, first = 0;
   ObjectList listP;

   for(;;) {
      if(hashP->table[first]) break;
      first++;
   }

   listP = hashP->table[first];

   for(i=first+1; i < size_HT(hashP); i++)
      if(hashP->table[i]) {

         *(listP->tailP) = *(hashP->table[i]->headP->nextP);
         listP->tailP = hashP->table[i]->tailP;
         listP->listlength += size_LL(hashP->table[i]);
        delete_Link(hashP->table[i]->headP->nextP);
         delete_Link(hashP->table[i]->headP);
         /* delete_Link(hashP->table[i]->headP->nextP); */
         free(hashP->table[i]);

      }

   return listP;
}

static void addto_Objects(ObjectHash hashP, int next_label, Color pixel, int y_pos, int
x_pos)
{
   setKey_HT(hashP, hash_Object(next_label));
   addObject_HT( hashP, new_Object(next_label, pixel, y_pos, x_pos) );
}

static void update_Objects( ObjectHash hashP, int object_label, int y_pos, int x_pos)
{
   Object *obj;
   const char *fn = "update_Objects";

   setKey_HT(hashP,hash_Object(object_label));

   if(findObject_HT(hashP, match_Object, &object_label)) {
      obj = (Object*)getObject_HT(hashP);

      obj -> x_min = MIN(obj -> x_min, x_pos);
      obj -> x_max = MAX(obj -> x_max, x_pos);
      obj -> y_min = MIN(obj -> y_min, y_pos);
      obj -> y_max = MAX(obj -> y_max, y_pos);
   }
}


static void combine_Objects(ObjectHash hashP, int b, int c, int *xmin, int *xmax, int
*ymin, int *ymax)
```

```
{
   dlink *linkP;
   Object *bP, *cP;

   if (b < c)
   {
        int temp;
        temp = b;
        b = c;
        c = temp;
   }

   setKey_HT(hashP,hash_Object(b));
   findNextObject_HT(hashP, match_Object, &b);
   bP = (Object*)getNextObject_HT(hashP);
   removeNextObject_HT(hashP);

   setKey_HT(hashP,hash_Object(c));
   findObject_HT(hashP, match_Object, &c);
   cP = (Object*)getObject_HT(hashP);

   *xmin = bP->x_min;
   *xmax = bP->x_max;
   *ymin = bP->y_min;
   *ymax = bP->y_max;

   cP -> x_min = MIN(cP -> x_min, bP->x_min);
   cP -> x_max = MAX(cP -> x_max, bP->x_max);
   cP -> y_min = MIN(cP -> y_min, bP->y_min);
   cP -> y_max = MAX(cP -> y_max, bP->y_max);

   delete_Object(bP);

   recycleLabel(b);
}


static void makegraymap_Objects( ColorHistogram *chP )
{
   register int i;

   chP->histogram = (ColorHistObject *) malloc(256*sizeof(ColorHistObject));

   for(i=0; i < 256; i++)
      assign_Color(chP->histogram[i].pixel, i, i, i);
}


static Matrix *color2index_Image(ColorHistogram *chP, Image *imageP )
{
   Matrix *matrixP;
   Color pixel;
   ColorHashTable *chtP;
   unsigned rows, cols;
   register int i;
   byte *mP, *rP, *gP, *bP;
```

```
    if(chP->no_of_colors > 256) {
       return NULL;
    }

    rows = getNoOfRows_Image(imageP);
    cols = getNoOfCols_Image(imageP);
    rP = (unsigned char*)getRow_Image(imageP, 0, RED);
    gP = (unsigned char*)getRow_Image(imageP, 0, GRN);
    bP = (unsigned char*)getRow_Image(imageP, 0, BLU);

    matrixP = new_Matrix(rows, cols, CVIP_BYTE, REAL);
    mP = (unsigned char*)getRow_Matrix(matrixP,0);

    chtP = hist2Hash_ColorHT(chP);

    for(i=0; i < rows*cols; i++, mP++, rP++, gP++, bP++) {

       assign_Color(pixel, *rP, *gP, *bP);

       if( (*mP = lookUpColor_ColorHT( chtP, pixel )) == -1 )
          return NULL;
    }
    delete_ColorHT(chtP);

    return matrixP;


static void initLabelStack(void)

       label_stackP = new_Stack();
       label_count=1;


static int getNextLabel(void)
{
       int next_label, *labelP;

       if(isempty_Stack(label_stackP))
             next_label = label_count++;
       else {
             labelP = (int*)pop_Stack(label_stackP);
             next_label = *labelP;
             free(labelP);
       }

       return next_label;
}

static void recycleLabel(int label)
{
       int *labelP = (int *) malloc(sizeof(int));
       *labelP = label;
       push_Stack(label_stackP, labelP);
}
```

```
// Connectedness.  Used in ImageThinning function.
int cconc(int inb[9])
{
      int icn = 0;

      for(int i=0; i<8; i+=2)
      {
            if(inb[i]==0)
                  if(inb[i+1] == 255 || inb[i+2] == 255)
                        icn++;
      }
      return icn;
}

// Performs thinning of a binary image
Image* ImageThinning(Image *img)
{
      // Copy original image
      Image *newimg = duplicate_Image(img);
      if(!newimg)
            AfxMessageBox("Not enough memory!");
      unsigned char **pix = (unsigned char **)newimg->image_ptr[0]->rptr;
      int m = 100;  int ir = 1, ia[9], ic[9];

            int numrows = img->image_ptr[0]->rows;
            int numcols = img->image_ptr[0]->cols;


      while(ir!=0)
      {
            ir=0;

            for(int iy=0; iy < numrows; iy++)
            {
                  for(int ix=0; ix < numcols; ix++)
                  {
                        if(pix[iy][ix] != 255)
                              continue;

                        if(ix == numcols-1)
                              ia[0] = 0;
                        else
                              ia[0] = pix[iy  ][ix+1];
                        if(iy == 0 || ix == numcols-1)
                              ia[1] = 0;
                        else
                              ia[1] = pix[iy-1][ix+1];
                        if(iy == 0)
                              ia[2] = 0;
                        else
                              ia[2] = pix[iy-1][ix  ];
                        if(iy == 0 || ix == 0)
                              ia[3] = 0;
                        else
                              ia[3] = pix[iy-1][ix-1];
                        if(ix == 0)
                              ia[4] = 0;
```

```
else
      ia[4] = pix[iy  ][ix-1];
if(iy == numrows-1 || ix == 0)
      ia[5] = 0;
else
      ia[5] = pix[iy+1][ix-1];
if(iy == numrows-1)
      ia[6] = 0;
else
      ia[6] = pix[iy+1][ix  ];
if(iy == numrows-1 || ix == numcols-1)
      ia[7] = 0;
else
      ia[7] = pix[iy+1][ix+1];

for(int i=0; i < 8; i++)
{
      if(ia[i] == m)
      {
            ia[i] = 255; ic[i] = 0;
      }
      else
      {
            if(ia[i] < 255)
                  ia[i] = 0;
            ic[i] = ia[i];
      }
}  // for
ia[8] = ia[0];
ic[8] = ic[0];
if(ia[0]+ia[2]+ia[4]+ia[6]==255*4)
      continue;

int iv, iw;
for(i=0, iv=0, iw=0; i<8; i++)
{
      if(ia[i]==255)
            iv++;
      if(ic[i]==255)
            iw++;
}
if(iv<=1)
      continue;
if(iw==0)
      continue;
if(cconc(ia)!=1)
      continue;

int temppix = (iy == 0) ? 0 : pix[iy-1][ix];
if(temppix == m)
{
      ia[2] = 0;
      if(cconc(ia) != 1)
            continue;
      ia[2] = 255;
}
temppix = (ix == 0) ? 0 : pix[iy][ix-1];
```

```
                        if(temppix == m)
                        {
                                ia[4]=0;
                                if(cconc(ia)!=1)
                                        continue;
                                ia[4] = 255;
                        }
                        pix[iy][ix] = m;
                        ir++;
                } // for ix
        } // for iy
        m++;
} // while
for(int iy=0; iy<img->image_ptr[0]->rows; iy++)
{
        for(int ix=0; ix<img->image_ptr[0]->cols; ix++)
        {
                if(pix[iy][ix] < 255)
                        pix[iy][ix] = 0;
        }
}
return newimg;
```